MOBPRO - Mobile Programming
Zusammenfassung FS 2019

Maurin D. Thalmann

11. März 2019

Inhaltsverzeichnis

1 Android 1 - Grundlagen

Informationen zur Androidprogrammierung können stets dem Android Developer Guide entnommen werden unter: *developer.android.com* Apps sollen grundsätzlich gegen das aktuellste API entwickelt werden, aktuell API Level 28 Android 9 „Pie“. Im Gradle-Build-Skript werden deshalb folgende SDK-Versionen festgehalten:

minSdkVersion Mindestanforderung an die SDK, Minimum-Version
targetSdkVersion Ziel-SDK-Version, auf welcher die App lauffähig sein soll
compileSdkVersion Version mit welcher die App (APK) erstellt wird, meist gleich der Target-Version

ART (Android Runtime) verwaltet Applikationen bzw. deren einzelne Komponenten:

- Komponente kann andere Komponente mit Intent-Mechanismus aufrufen
- Komponenten müssen beim System registriert werden (teilweise mit Rechten = Privileges)
- System verwaltet Lebenszyklus von Komponenten: Gestartet, Pausiert, Aktiv, Gestoppt, etc.

1.1 Komponenten

Applikationen sind aus Komponenten aufgebaut, die App verwendet dabei eigene Komponenten (min. eine) oder Komponenten von anderen, existierenden Applikationen.

Name	***Beschreibung***
Activity	UI-Komponente, entspricht typischerweise einem Bildschirm
Service	Komponente ohne UI, Dienst läuft typischerweise im Hintergrund
Broadcast Receiver	Event-Handler, welche auf App-interne oder systemweite Broadcast-Nachrichten reagieren
Content Provider	Komponente, welche Datenaustausch zwischen versch. Applikationen ermöglicht

Activity entspricht einem Bildschirm, stellt UI-Widgets dar, reagiert auf Benutzer-Eingabe & -Ereignisse. Eine App besteht meist aus mehreren Activities / Bildschirmen, die auf einem „Stack“ liegen.
Basisklasse: *android.app.Activity*
Service läuft typischerweise im Hintergrund für unbeschränkte Zeit, hat keine graphische Benutzerschnittstelle (UI), ein UI für ein Service wird immer von einer Activity dargestellt.
Basisklasse: *android.app.Service*
Broadcast Receiver ist eine Komponente, welche Broadcast-Nachrichten empfängt und darauf reagiert. Viele Broadcasts stammen vom System (Neue Zeitzone, Akku fast leer,...), App kann aber auch interne Broadcasts versenden.
Basisklasse: *android.content.BroadcastReceiver*
Content Provider ist die einzige *direkte* Möglichkeit zum Datenaustausch zwischen Android-Apps. Bieten Standard-API für Suchen, Löschen, Aktualisieren und Einfügen von Daten.
Basisklasse: *android.content.ContentProvider*

1.2 Das Android-Manifest

AndroidManifest.xml dient dazu, alle Komponenten einer Applikation dem System bekannt zu geben. Es enthält Informationen über Komponenten der Applikation, statische Rechte (Privileges), Liste mit Erlaubnissen (Permissions), ggf. Einschränkungen für Aufrufe (Intent-Filter). Es beschreibt die statischen Eigenschaften einer Applikation, beispielsweise:
(Diese Infos werden bei der App-Installation im System registriert, zusätzliche Infos (Version, ID, etc.) befinden sich im Gradle-Build-Skript (können build-abhängig sein))

- Java-Package-Name
- Benötigte Rechte (Internet, Kontakte, usw.)
- Deklaration der Komponenten
  - Activities, Services, Broadcast Receivers, Content Providers
  - Name (+ Basis-Package = Java Klasse)
  - Anforderungen für Aufruf (Intent) für A, S, BR
  - Format der gelieferten Daten für CP

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="ch.hslu.mobpro.hellohslu">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity
            android:name=".MainActivity"
            android:label="@string/app_name"
            android:theme="@style/AppTheme.NoActionBar">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Abbildung 1: Beispiel eines Android-Manifests

1.3 Activities & Aufruf mit Intents

Zwischen Komponenten herrscht das Prinzip der losen Kopplung:

- Komponenten rufen andere Komponenten über Intents (= Nachrichten) auf
- Offene Kommunikation: Sender weiss nicht ob Empfänger existiert
- Parameterübergabe als Strings (untypisiert)
- Parameter: von Empfänger geprüft, geparst & interpretiert (oder ignoriert)
- Keine expliziten Abhängigkeiten $\rightarrow$ Robuste Systemarchitektur

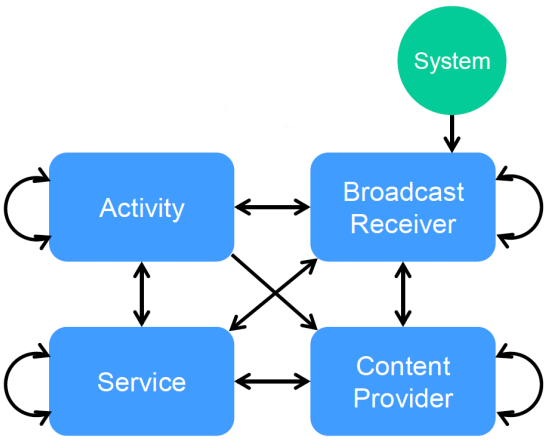


Abbildung 2: Kommunikation zwischen Komponenten mit Intents

Intents werden benutzt, um Komponenten zu benachrichtigen oder um Kontrolle zu übergeben. Es gibt folgende zwei Arten von Intents:
Explizite Intents adressieren eine Komponente direkt
Implizite Intents beschreiben einen geeigneten Empfänger
WICHTIG: Activities müssen immer im Manifest deklariert werden, da sie sonst nicht als „public“ gelten und eine Exception schmeissen. Das geht auch ganz einfach folgendermassen im Manifest unter „application“:

```
<activity android:name=".Sender" />
<activity android:name=".Receiver" />
```

1.3.1 Beispielaufruf Expliziter Intent

Sender Activity:

```
public void onClickSendBtn(final View btn) {
   Intent intent = new Intent(this, Receiver.class);
   // Receiver.class ist hier der explizite Empfaenger
   intent.putExtra("msg", "Hello World!");
   startActivity(intent);
}
```

Receiver Activity:

```
public void onCreate(Bundle savedInstanceState) {
   // ...
   Intent intent = getIntent();
   String msg = intent.getExtras().getString("msg");
   displayMessage(msg);
}
```

1.3.2 Beispielaufruf Impliziter Intent

Sender Activity:

```
Intent browserCall = new Intent();
browserCall.setAction(Intent.ACTION_VIEW);
browserCall.setData(Uri.parse("http://www.hslu.ch"));
startActivity(browserCall);
```

ACTION_VIEW ist hierbei kein expliziter Empfängertyp, sondern nur eine gewünschte Aktion. Die mitgegebene URL wird auch ein *Call Parameter* genannt. Gesucht ist in diesem Fall eine Komponente, welche eine URL anzeigen/verwenden kann.

1.4 Activities & Subactivities

Activity Back Stack: Activities liegen aufeinander wie ein Stapel Karten, neuste Activity zuoberst und in der Regel ist nur diese sichtbar (Durch Transparenz sind hier Ausnahmen möglich). Durch „back“ oder „finish“ wird die oberste Karte entfernt und man kehrt zur zweitletzten Activity zurück. Mehrere Instanzen derselben Activity wären mehrere solche Karten, das Verhalten kann jedoch konfiguriert werden (z.Bsp. maximal eine Instant, mehrere Activities öffnen, etc.)
(Sub-)Activities und Rückgabewerte: Eine Activity kann Rückgabewerte einer anderen (Sub-)Activity erhalten.

```
// 1. Aufruf der SubActivity mit
startActivityForResult(intent, requestId)

// 2. SubActivity setzt am Ende Resultat mit
setResult(resultCode, intent) // intent als Wrapper fuer Rueckgabewerte

// 3. SubActiity beendet sich mit
finish()

// 4. Nach Beendung der SubActivity wird folgendes im Aufrufer aufgerufen:
onActivityResult(requestId, resultCode, intent)
// resultCode: RESULT_OK, RESULT_CANCELLED
```

1.5 Lebenszyklus & Zustände von Applikationen/Activities

Das System kann Applikationen bei knappem Speicher ohne Vorwarnung terminieren (nur Activities im Hintergrund, dies geschieht unbemerkt vom User, die App wird bei Zurücknavigation wiederhergestellt). Eine Applikation kann ihren Lebenszyklus demnach nicht kontrollieren und muss in der Lage sein, ihren Zustand speichern und wieder laden zu können. Applikationen durchlaufen mehrere Zustände in ihrem Lebenszyklus, Zustandsübergänge rufen Callback-Methoden auf (welche von uns überschrieben werden können.

Activity-Zustände:

Zustand	Beschreibung
Running	Die Activity ist im Vordergrund auf dem Bildschirm (zuoberst auf dem Activity-Stack für die aktuelle Aufgabe).
Paused	Die Activity hat den Fokus verloren, ist aber immer noch sichtbar für den Benutzer.
Stopped	Die Activity ist komplett verdeckt von einer andern Activity. Der Zustand der Activity bleibt jedoch erhalten.

1.5.1 Lifecycle einer Applikation

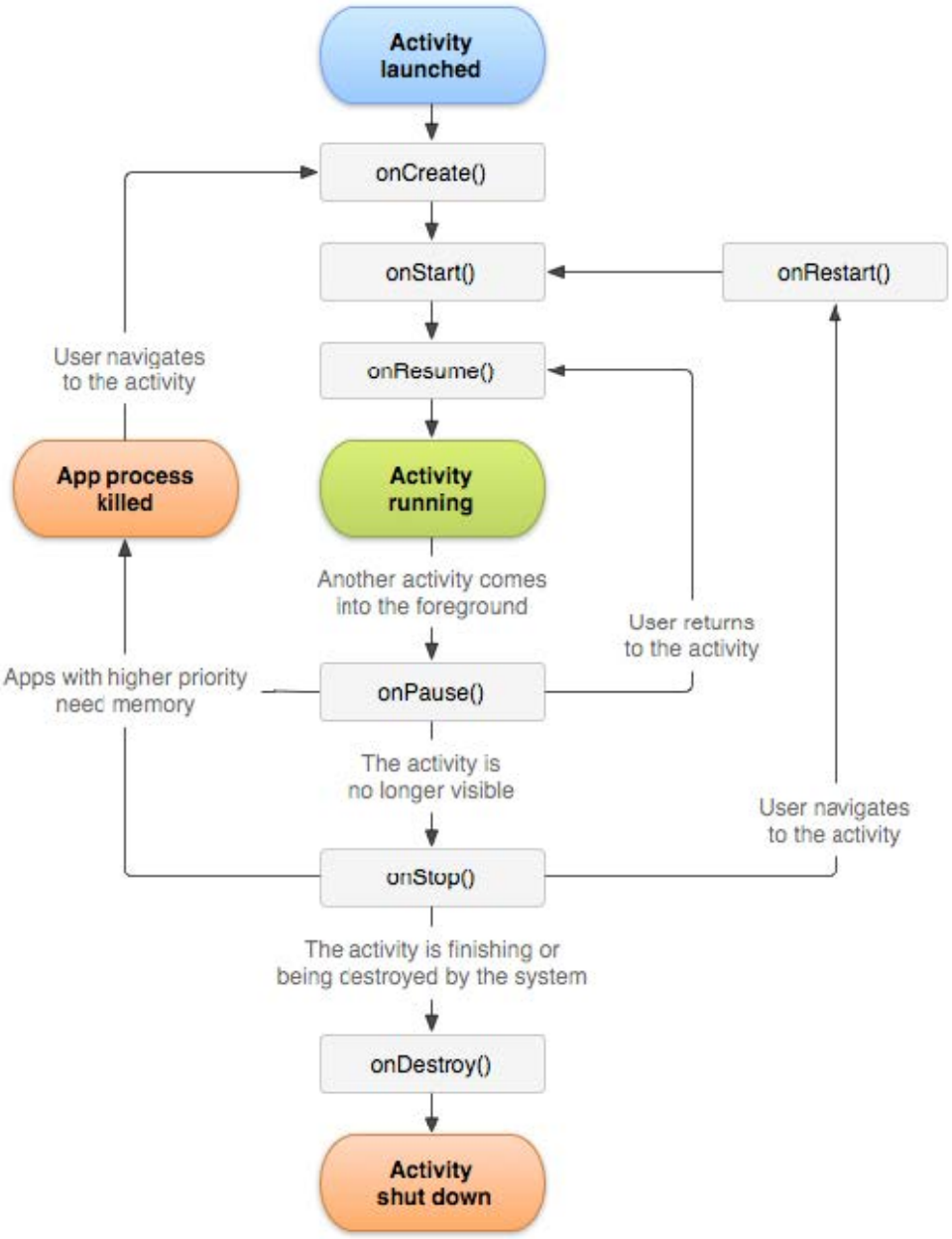


Abbildung 3: Lifecycle einer Applikation

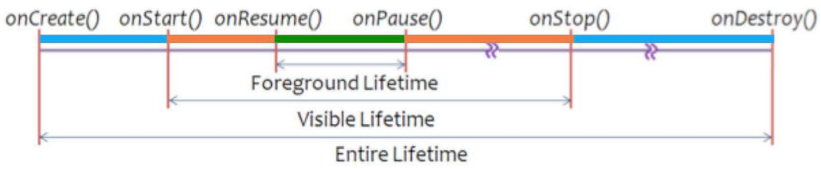


Abbildung 4: Lebenszeiten der einzelnen App-Zustände

1.6 Charakterisierung einer Activity

- Muss im Manifest deklariert werden
- GUI-Controller
  - Repräsentiert eine Applikations-/Bildschirmseite
  - Definiert Seitenlayout und GUI-Komponenten
  - Kann aus Fragmenten (= „Sub-Activities“) aufgebaut sein
  - Reagiert auf Benutzereingaben
  - Beinhaltet Applikationslogik für dargestellte Seite

Beispiel einer Activity:

```
public class Demo extends Activity {
  // Called when the Activity is first created
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main); // Definiert Layout und UI
  }
}
```

1.6.1 Zustandsänderung - Hook-Methoden

Das System benachrichtigt Activities durch Aufruf einer der folgenden Methoden der Klasse *Activity*:

- void onCreate(Bundle savedInstanceState)
- void onStart() / void onRestart()
- void onResume()
- void onPause() → *bspw. Animation stoppen*
- void onStop()
- void onDestroy() → *bspw. Ressourcen freigeben*

Durch das Überschreiben dieser Methoden können wir uns in den Lebenszyklus einklinken. Immer **super()** aufrufen, sonst wirft es eine Exception.

1.7 Android - Hinter den Kulissen

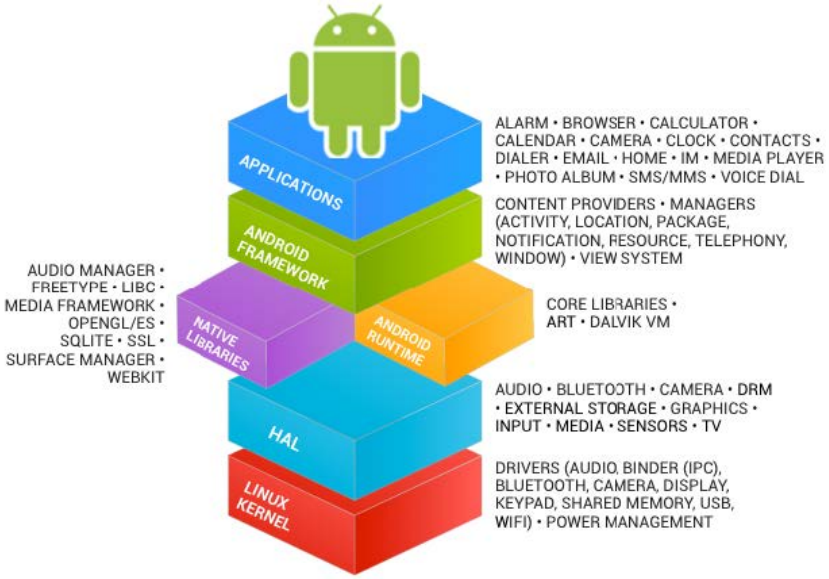


Abbildung 5: Der Android-Stack

- **Linux-Kernel:** OS, FS, Security, Drivers, ...
- **HAL (Hardware Abstraction Layer):** Camera-, Sensor-, ... Abstraktion
- **ART** (Android Runtime)
  - Jede App in eigenem Prozess
  - Optimiert für mehrere JVM auf low-memory Geräten
  - Eigenes Bytecode-Format (Crosscompiling)
  - JIT und AOT Support
- **Native C/C++ Libriaries:** Zugriff via Android NDK
- **Android Framework:** Android Java API
- **Applications:** System- und eigene Apps

1.7.1 Android-Security-Konzept

Sandbox-Konzept:

- Jede laufende Android-Anwendung hat seinen eigenen Prozess, Benutzer, ART-Instanz, Heap und Dateisystembereich → jedes App hat eigenen Linux-User
- Das Berechtigungssystem von Linux ist Benutzer-basiert, es betrifft deshalb sowohl den Speicherzugriff wie auch das Dateisystem.
- Anwendungen signieren: erschwert Code-Manipulationen und erlaubt das Teilen einer Sandbox bei gleicher sharedUser-ID
- Berechtigungen werden im Manifest deklariert, kontrollierte Öffnung der Sandbox-Restriktionen

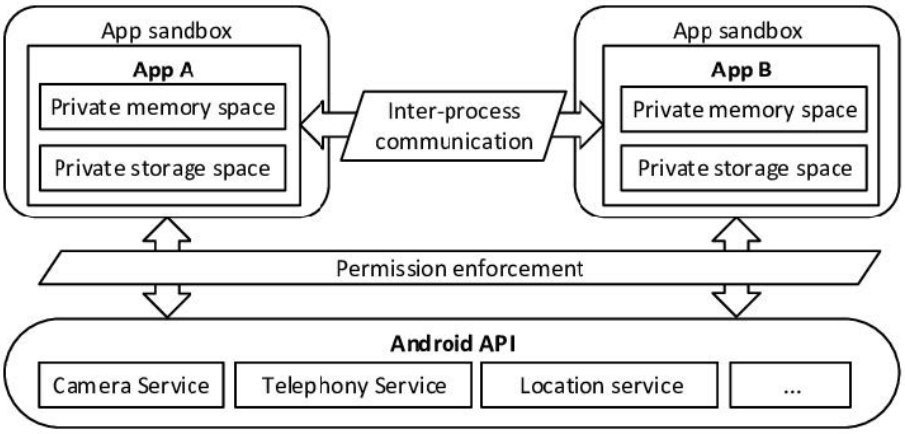


Abbildung 6: Android Security-Modell

2 Android 2 - Benutzerschnittstellen

2.1 GUI einer Activity

GUI wird als XML definiert, der Name resultiert in einer Konstante: ***R****.layout.xxx*. Diese wird im *onCreate()* einer Activity mit *setContentView()* angegeben.

```
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        android:padding="@dimen/padding">

        <TextView
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:paddingBottom="@dimen/padding"
            android:text="@string/main_title"
            android:textSize="@dimen/textSizeTitle" />

        <TextView
            android:layout_width="match_parent"
```

Abbildung 7: Beispiel eines XML für ein Layout

Je nach Layout müssen die Elemente unterschiedlich konfiguriert werden, was bei der Arbeit mit dem Layout-Editor nicht offensichtlich, aber trotzdem gut zu wissen ist.
Ein Android-UI ist hierarchisch aufgebaut und besteht aus **ViewGroups** (Cointainer für Views oder weitere ViewGroups, angeordnet durch Layout) und **Views** (Widgets). Sollte auf unterschiedlichen Bildschirmgrössen gleich aussehen (Elemente deshalb **relativ** und nicht absolut positionieren)

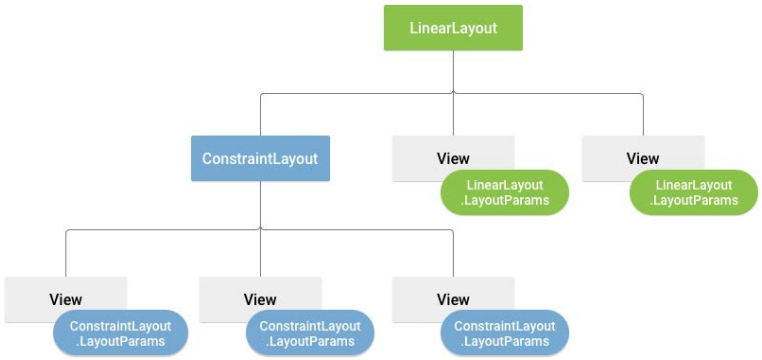


Abbildung 8: Layout-Varianten bei Android

Schachtelung möglich, aber nicht effizient, wenn möglich immer das Constraint-Layout verwenden. Layouts spezifiziert man auf zwei verschiedene Arten:

- **Statisch / Deklarativ (XML)**
- *Grundsätzlich in MOBPRO verwendet, bietet viele Vorteile (Deklarativ, weniger umständlich als Code, Struktur eminent, Umformungen ohne Rekompilierung möglich...)*
  - Deklarative Beschreibung des GUI als Komponentenbaum
  - XML-Datei unter *res/layout*
  - Referenzen auf Bilder/Texte/etc.
  - Typischerweise ein XML pro Activity
- **Dynamisch (in Java)**
- *Jedes XML hat eine korrespondierende Java-Klasse, XML → Java = Inflating*
  - Aufbau und Definition des GUI im Java-Code
  - Normalerweise nicht nötig: die meisten GUIs haben fixe Struktur
  - Änderung von Eigenschaften während Laufzeit ist normal (Bsp. Visibility, Ausblenden einer View, wenn nicht benötigt)

2.2 XML-Layout

- Jedes Layout ist ein eigenes XML-File
  - Root-Element = View oder ViewGroup
  - Kann Standard- oder eigene View-Klassen enthalten
- XML können mit Inflater „aufgeblasen“ bzw. instanziiert werden, damit eigene wiederverwendbare Komponenten/Templates/Prototypen erzeugt werden können
- Innere Elemente können unterhalb eines Parents via View-ID referenziert werden (*findViewById()*)
- Debugging mit dem Layout-Inspector

2.2.1 Constraint-Layout

- Erstellung von komplexen Layouts, ohne zu schachteln
- Elemente werden relativ mit Bedingungen platziert
  - zu anderen Elementen
  - zum Parent-Container
  - Element-Chains (spread/pack)
- Layout-Hilfen (Hilfslinien, Barriers)

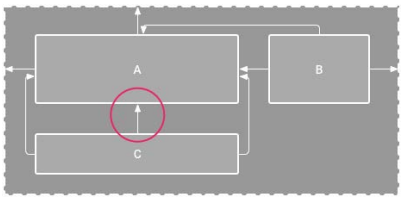


Abbildung 9: Constraint Layout

2.2.2 LinearLayout

- Reiht Elemente neben-/untereinander auf
  - kann geschachtelt werden, um Zeilen/Spalten zu formen (nicht zu tief, sonst schlechte Performanz
- Eigenschaften:
  (orientation, gravity, weigthSum, etc.)
- Layout-Parameter für Children
  - layout_width, layout_height
  - layout_margin...
  - layout_weight, layout_gravity

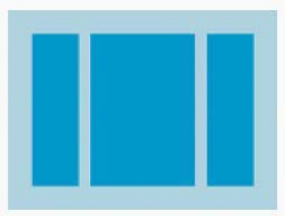


Abbildung 10: LinearLayout

Warum nutzt man trotzdem noch LinearLayout?

- Nach wie vor einfachste Lösung für Button- oder Action-Bars („flow semantik“) und einfache Screens
- Kaum Konfiguration nötig, robust
- Für scrollbare Listen mit dynamischer Anzahl Elemente besser *ListView* verwenden (siehe Adapter-Views)
- Einsatz mit Bedacht durchaus sinnvoll

Es gibt noch die **ScrollView**, deren Nutzung vertikales Scrollen bei zu grossen Layouts erlaubt, sie kann jedoch nur ein Kind haben und enthält typischerweise das Top-Level-Layout einer Bildschirmseite.

Pixalangaben *(Typischerweise werden Angaben in dp verwendet, ausser sp bei Schriftgrössen.)*

- **dp - density-independent:**
  Passen sich der physischen Dichte des Screens an, dp passen sich gegenüber den realen Dimensionen eines Screens und dessen Verhältnisse an.
- **sp - scale-independent:**
  Ähnlich der dp-Einheit, passt sich jedoch der Schriftskalierung des Nutzers an.
- **px - Pixels:**
  Passen sich der Anzahl Pixel eines Bildschirms an, deren Nutzung wird nicht empfohlen.

2.3 Ressourcen, Konfigurationen und Internationalisierung

Ressourcen sind alle Nicht-Java-Teile einer Applikation und sind im */res*-Verzeichnis abgelegt, sogenannte ausgelagerte Konstanten-Definitionen. Sie werden im Layout und Java-Code über die ***automatisch generierte R-Klasse*** mit ID-Konstanten (int) referenziert. Kontextabhängige Ressourcen sind möglich z.Bsp. für Sprache, Gerätetyp, Orientierung, ...
Beispiele: Strings, Styles, Colors, Dimensionen, Bilder (drawables), Layouts (portrait, landscape), Array-Werte (z.Bsp. für Spinner) und Menü-Items

```
<TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginBottom="@dimen/marginBottom"
    android:background="@color/sectionBackground"
    android:padding="@dimen/padding"
    android:text="@string/main_section1"
    android:textColor="@color/sectionText" />
```

(a) Referenz in XML mit @

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
```

(b) Referenz in Code über `R`-Klasse, diese wird beim Build automatisch generiert

Für verschiedene Systemkonfigurationen benötigt es unterschiedliche Ausprägungen einer Ressource, beispielsweise:

- **Internationalisierung**: komplette/teilweise Übersetzung, für diese werden unterschiedliche Ordner je nach Land/Sprache und seperate .xml angelegt
- **Auflösungsklassen**: `ldpi` (120dpi), `mdpi` (160dpi), `hdpi` (240dpi), `xhdpi` (320dpi)
- **Orientierung** des Displays: landscape / portrait
- Verschiedene **HW-Modelle**: HTC, Samsung, Sony, LG, ...

Default-Verzeichnisse sind innerhalb von `res/` angelegt: drawable, layout, menu, values, ...
Bei spezifischen Konfigurationen werden meist Kopien der Default-Verzeichnisse/Ordner mit einem Suffix angelegt, bspw. `res/strings-de-rCH`, in welchen dann die Ressourcen (XML) erneut angelegt werden.

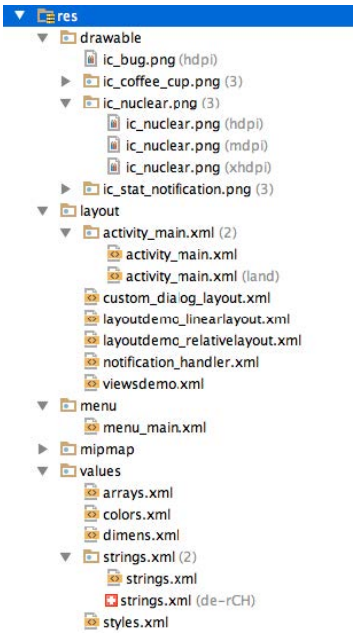


Abbildung 12: Beispiel der Default-Ressourcen

2.4 UI-Event-Handling

- Jedes View-Element hat eine entsprechende Java-Klasse (auch View-Groups!)
  → Layout könnte auch dynamisch in Java programmiert werden
- APIs der einzelnen View-Klassen sind hier oder unter „Nützliche Links“ genauer beschrieben

```
<TextView
    android:id="@+id/message_label"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
// Show message on dedicated text view
private void displayMessage(String message) {
    TextView label = (TextView)findViewById(R.id.message_label);
    label.setText(message);
}
```

Abbildung 13: ID im Layout erfassen und Referenz im Code

2.4.1 GUI-Events

- **Observer/Listener**: einen Listener für ein entsprechendes Event bei der View registrieren, bspw. bei `Button myButton`:
  `myButton.setOnClickListener(listener)`
- verschiedenste Event- und Listener-Typen:
  `OnClickListener, OnLongClickListener, OnKeyListener, OnTouchListener, OnDragListener, ...`
  → `public static` Interfaces der Klasse `View`

Ziel: Auf Klick-Event eines Buttons reagieren

- Button muss eine ID haben im layout.xml
- Registrierungs eines Listeners an die View (Button) im Code:

```
Button button = (Button) findViewById(R.id.question_button_done);
button.setOnClickListener(new OnClickListener() {
  @Override
  public void onClick(View v) {
    // handler code
    buttonClicked();
  }
});
```

onClick-Event-Registrierung in XML

```
<Button
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:onClick="increaseInternalCounter"
    android:layout_marginBottom="@dimen/marginBottom"
    android:text="@string/main_increaseInternalCounter" />
```

Abbildung 14: Definition onClick-Handler im Layout → so nur für OnClick-Events

```
// Implementierung OnClick-Handler-Methode in der Activity
public void increaseInternalCounter(View button){
  // ... handler code ...
}
```

2.4.2 Exkurs: Data Binding

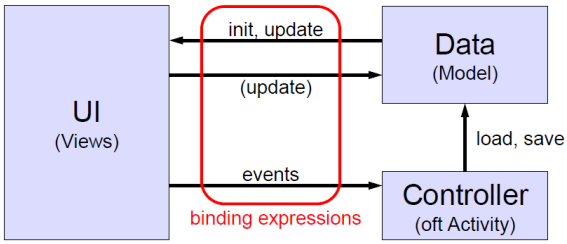


Abbildung 15: Modell für Data Binding

Data Binding: separiert UI und Daten, synchronisiert UI mit Daten (1-, resp. 2-way-binding), verwendet «binding expressions» mit @.. Syntax im Layout-File, um View-Attribute zu initialisieren. Anbei ein Beispiel (auskommentiert):

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android">
  <data>
    <variable name="model" type="org.example.MyModel"/>
  </data> // Definition der Layout-Variablen
  <LinearLayout ...>
    <Button
      android:id="@+id/button"
      ...
      android:enabled="@{model.user.role == `admin`}"
      android:text="@{model.buttonText}" // Data Binding (1-way)
      ...
      android:onClick="@{() -> model.increaseClickCount()}" /> // Event Binding
    <EditText
      android:id="@+id/input"
      ...
      android:text="@={model.inputText}"/> // Data Binding (2-way)
  </LinearLayout>
</layout>

protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  ActivityMainBinding binding = DataBindingUtil.setContentView(...);
  model = new MainModel();
  model.load();
  binding.setModel(model);
  // Binden der Layout-Daten auf effektive Daten
  // z.B. ViewModel mit Observables
}
```

2.5 Options-Menü

- Android-Apps können oben rechts ein Menü mit Optionen anbieten
- Erzeugung durch Aufruf *Hook* in der Activity-Klasse:
  `onCreateOptionsMenu(Menu menu)`
  - Hier kann ein Menü mit Einträgen bestückt werden
  - `MenuInflater` + XML benutzen oder Java oder beides
- Beim Klick auf Eintrag Aufruf eines anderen Hooks:
  `onOptionsItemSelected(MenuItem item)`

Für ein Options-Menü muss eine .xml-Datei (Bsp. main_menu.xml) im Ordner `res/menu` angelegt werden. Danach werden Informationen folgendermassen eingetragen:

```xml
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools" tools:context=".MainActivity">
    <item
        android:id="@+id/main_menu_finish"
        android:title="@string/menu_finish">
    </item>
    <item
        android:id="@+id/main_menu_startAllViews"
        android:title="@string/menu_startViewsDemo">
    </item>
</menu>
```

(a) Menü und Items in XML definieren

```java
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    super.onCreateOptionsMenu(menu);
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.menu_main, menu);
    return true;
}
```

(b) Menü mit `MenuInflater` aufblasen

Um bspw. einen String in einem Menüpunkt einzufügen, gibt es drei verschiedene Möglichkeiten:

```java
menu.add(Menu.NONE, 239, Menu.NONE, "Menu Item 1");
menu.add(Menu.NONE, 333, Menu.NONE, getString(R.string.menu_mail));
menu.add(Menu.NONE, 923, Menu.NONE, R.string.menu_server);
```

Abbildung 17: Möglichkeiten zum Einlesen eines Strings

```java
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if (super.onOptionsItemSelected(item)) {
        return true; // handled by super implementation
    }
    switch (item.getItemId()) {
        case R.id.main_menu_finish:
```

Abbildung 18: Event-Handling: Selektierung

2.6 Adapter-Views

Behandelt wird hier nur das synchrone Laden von kleinen/schnellen Datenquellen, für asynchrones Laden von langsamen/grossen Datenquellen konsultiere Doku über ***Loaders****.*

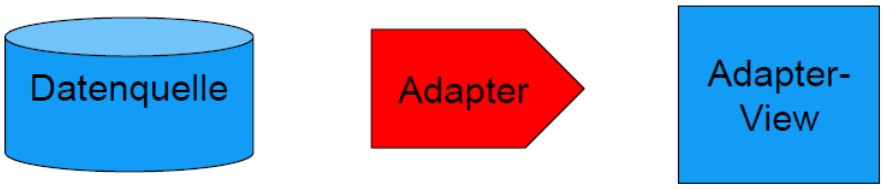


Abbildung 19: Aufgabe des Adapters

- Adapter → Verbindung zwischen Datenquelle und GUI
- Zapft *Datenquelle* an und beliefert *AdapterView*
- Erzeugt (Sub-)Views pro gefundenes Datenelement
- Transformiert Daten ggf. in benötigtes Zielformat
- Datenquellen:
  String-Array, String-Liste, Bilder, Datenbank, ...

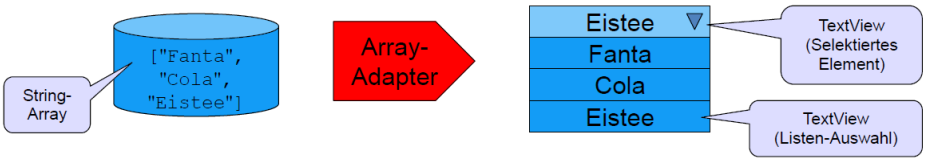


Abbildung 20: Beispiel eines ArrayAdapter

- Bindet irgend ein Array oder Liste mit beliebig getypeten Elementen an irgend eine AdapterView
- Für jedes Daten-Element wird eine SubView erzeugt
- **Default**: Erstellt `TextView` mit `element.toString()`-Wert

```
String[] myArray = new String[]{"Fanta", "Cola", "Eistee" };
ArrayAdapter<String> adapter =
        new ArrayAdapter<String>(this, android.R.layout.simple_list_item_checked, myArray);
this.setListAdapter(adapter);
```

Abbildung 21: Beispiel einer AdapterView

2.6.1 AdapterViews & ListActivity

- **AdapterViews**: spezielle View-Klassen
  - Sind für Zusammenarbeit mit Adaptern optimiert
    (Bsp.`ListView, GridView, Gallery, Spinner, Stack,...`)
  - Füllen Teile von sich mit von Adaptern erzeugten Views
  - Leiten ab von `android.widget.AdapterView<T extends android.widget.Adapter>`
- Spezielle Activity: **ListActivity**
  - Vordefiniertes Layout (enthält eine `ListView`, kein XML nötig)
  - Vordefinierte Callbacks (bei Auswahl einer List-Entry)
  - Bietet Zugriff auf aktuelle Selektion / Datenposition

2.6.2 android.widget.Spinner

- ComboBox oder DropDown-List genannt (weitere Alternative: AutoCompleteTextView)
- Zeigt ein ausgewähltes Element, bei Klick erscheint ein Auswahlmenü
- 2 Varianten, um Daten auf Spinner zu setzen:
  - Im Code mit Adapter:
    `spinner.setAdapter(myAdapter)`
  - Im XML mit Angabe einer String-Array-ID:
    `android:entries=„@array/spinnerValues“`
- Listener setzen für Behandlung der Auswahl:
  `spinner.setOnItemSelectedListener(...)`

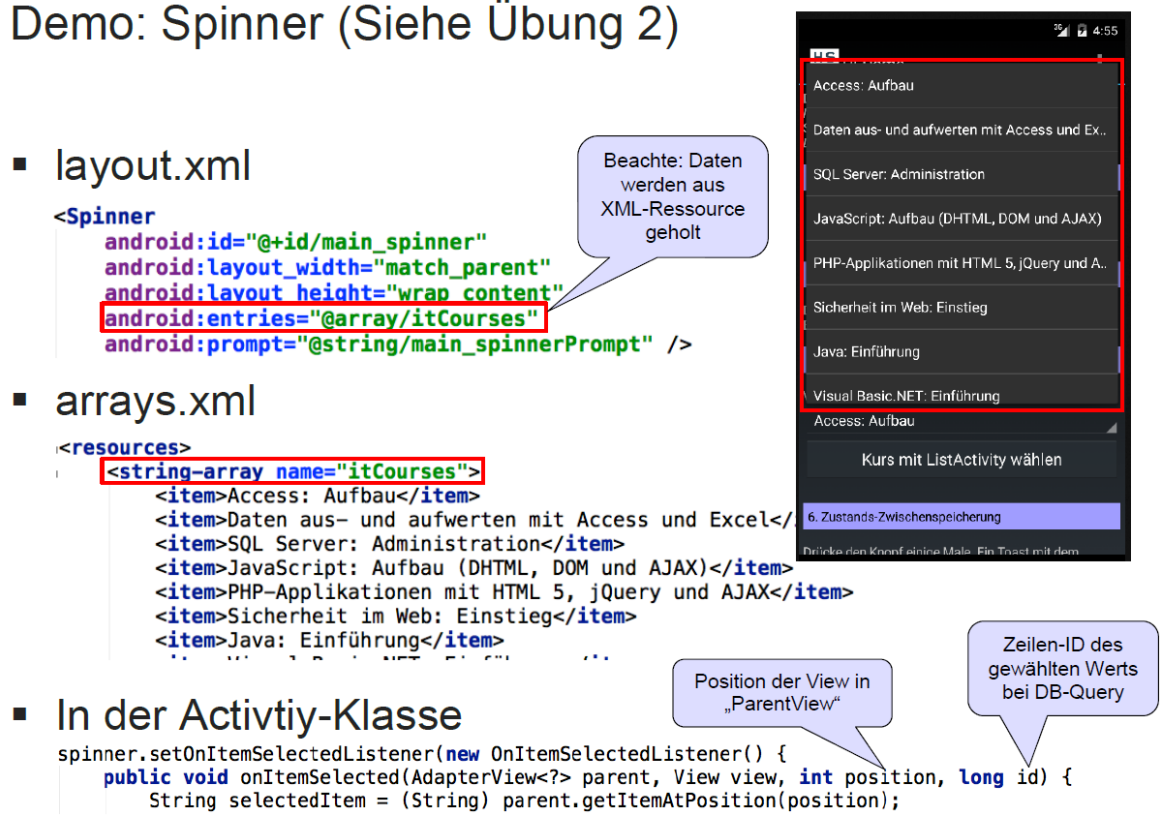


Abbildung 22: Übungs-Demo aus der Vorlesung SW02 - Spinner

2.6.3 android.widget.ListView

- Liste von Views/Items, die zur Auswahl stehen
- Braucht viel Platz! Meist wird ihr der ganze Bildschirm zugeteilt
- i.d.R. zusammen mit `ListActivity` verwendet, Verwendung:
  1. Navigiere zu eigener ListActivity
  2. Auswahl → Resultat setzen → finish
  3. Auswertung des Rückgabewert im Caller
- Konzeptionell identisch zum Spinner, jedoch andere Darstellung auf UI
  - Verwendungsentscheid:
    - Kurze Liste → Spinner
    - (Sehr) lange Listen → ListView / ListActivity
    - Kennt der User die möglichen Auswahlwerte → AutoCompleteTextView
  - Adapter- / Datendefinition grundsätzlich bei beiden gleich (d.h. im Code oder durch XML-Array)
  - Auswahlmodus: `setChoiceMode(ListView.CHOICE_MODE_*` → Single- / Multiselection

2.6.4 android.app.ListActivity

- Spezielle Activity zur Darstellung einer ListView
- Vordefiniertes Layout (full-screen Liste)
  - `setContentView(...)` muss nicht aufgerufen werden
  - Aufruf i.d.R. mit `startActivityForResult(...)`
  - Vordefinierte vererbte Konfigurationsmethoden
    - `setListAdapter(adapter)` setzt Daten für die Liste
    - `getListView()` erlaubt Zugriff auf ListView-Instanz *(anstelle von findViewById(..) + Casten)*
- Callback bei der Auswahl
  - `onListItemClick(parentView, view, position, id)` Wird bei Auswahl aufgerufen (muss in Subklasse überschrieben werden, keine Listener-Registrierung nötig)

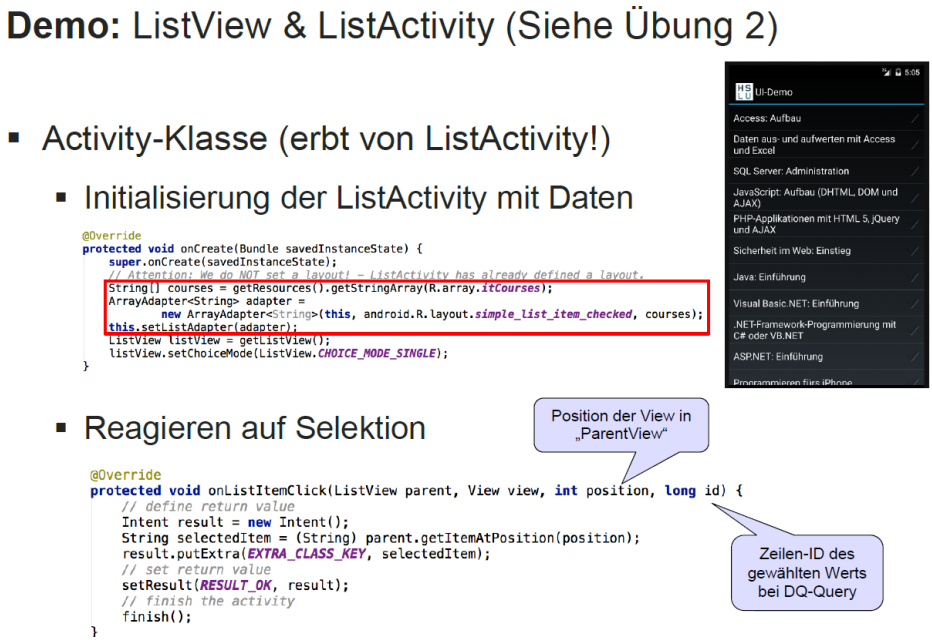


Abbildung 23: Übungs-Demo aus der Vorlesung SW02 - ListView / ListActivity

2.7 ViewModel - Konfigurationswechsel & temporäre Datenspeicherung

Bei jedem Konfigurationswechsel (z.B. Änderung Bildschirmorientation) wird die aktuelle Activity-Instanz zerstört und neu aufgebaut. Dabei besteht das Problem des **Zustandsverlusts**. Der Zustand alles Views mit einer ID (mit einigen Ausnahmen) wird automatisch gesichert und wiederhergestellt. Der **inhärente Zustand**, alles was nicht sichtbar und in Feldern gespeichert ist, geht jedoch verloren. Um entgegenzuwirken, kann ein **ViewModel** verwendet werden.

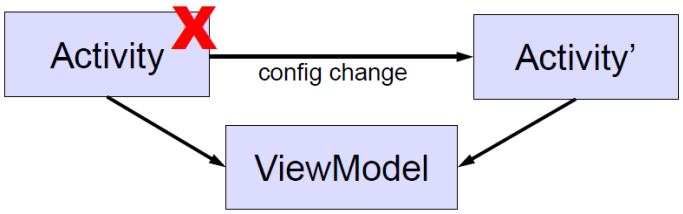


Abbildung 24: Position des ViewModels in der temp. Datenspeicherung

- Kapselt UI-Daten so, dass diese bei einer Konfigurationsänderung einer Activity in-memory erhalten bleiben *(Für den Fall eines App-Kills müssen Daten immer noch persistiert werden)*
- Lebensdauer mit der Activity gekoppelt
- Weniger Aufwand für Behandlung von Konfigurationsänderungen

Abbildung 25: Übungs-Demo aus der Vorlesung SW02 - ViewModel

2.8 Rückmeldungen an den Benutzer

2.8.1 Toast

- Kurze Rückmeldung (Popup) an den Benutzer, keine Interaktion möglich, verschwindet nach gewisser Zeit.
- Konfiguration: Text, Layout, Anzeigezeit (kurz/lang), Ort (gravity)
- Toasts mit eigenem Layout werden mit CustomToastView erstellt

Beispielcode zur Erstellung von Toasts:

```
// Default-Toast: Einzeiler
Toast.makeToast(getApplicationContext(), "Das ist..", Toast.LENGTH_LONG).show()
                                                    // LENGTH: Nur LONG oder SHORT
                // Kontext: meistens "this"

// Toast mit anderem Anzeigeort:
Context context = getApplicationContext();
Toast toast = Toast.makeText(context, "Toast links oben!", Toast.LENGTH_LONG);
toast.setGravity(Gravity.TOP|Gravity.START,0,0); // (x,y) Offset
toast.show()
```

2.8.2 Alert-Dialog

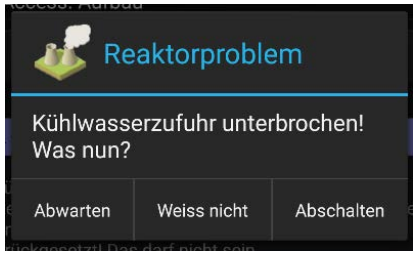


Abbildung 26: Beispiel eines Alert-Dialogs

- Fenster mit Interaktionsmöglichkeiten für den Benutzer
  - Information / Eingabe von Daten
  - Interaktion möglich
  - Buttons: positive, neutral, negative
- Vorteile
  - Kaum Einschränken in punkto Darstellung
  - Vorbereitet für die Anzeige von Daten
  - Verschwindet erst, wenn sie vom Benutzer quittiert wurde
- Konfiguration: Buttons, Titel, Icon, Nachricht
  Inhalt: Liste von Items oder eigene View

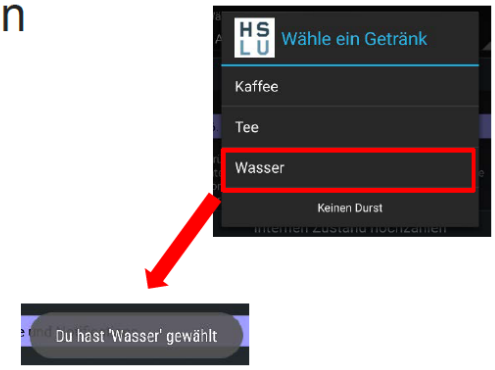
- Vorgehen beim Erstellen eines Alert-Dialog mit Builder-Muster
  1. Builder erstellen: `new AlertBuilder.Builder(this)`
  2. Builder konfigurieren:
     `setXXX` + Registrierung von `ClickListeners`
  3. Dialog erstellen: `Dialog dialog = builder.create()`
  4. Dialog anzeigen: `dialog.show()`
- Anzeige von Dialogen ist **immer asynchron!**
  Bei `show()` wird nicht gewartet, kein Rückgabewert
  → Behandlung von Benutzerselektion mit Listener

```
AlertDialog.Builder dialogBuilder = new AlertDialog.Builder(this);
dialogBuilder.setTitle("Reaktorproblem")
        .setIcon(R.drawable.ic_nuclear)
        .setMessage("Kühlwasserzufuhr unterbrochen!\nWas nun?")
        .setPositiveButton("Abschalten", new OnClickListener() {
            public void onClick(DialogInterface dialog, int which) {
                Toast.makeText(getApplicationContext(),
                        "Reaktor wird abgeschaltet...",
                        Toast.LENGTH_LONG).show();
            }
        }).setNeutralButton("Weiss nicht", new OnClickListener() {
            public void onClick(DialogInterface dialog, int which) {
                Toast.makeText(getApplicationContext(),
                        "Problem an Support weitergeleitet...",
                        Toast.LENGTH_SHORT).show();
            }
        }).setNegativeButton("Abwarten", new OnClickListener() {
            public void onClick(DialogInterface dialog, int which) {
                // do nothing
            }
        });
return dialogBuilder.create();
```

Abbildung 27: Beispiel eines AlertDialog aus Vorlesung

Alert-Dialog mit Auswahl-Daten

- Titel, Icon, usw. wie gehabt
- Neu: Daten (Array) setzten
  - Methode `setItems(...)`
    - Inkl. ClickListener
      - Toast mit Wahl anzeigen!



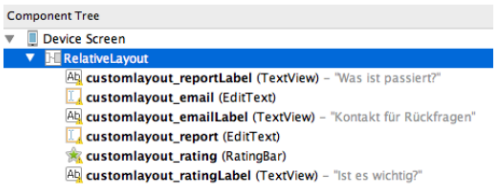

```
final String[] items = {"Kaffee", "Tee", "Wasser"};
AlertDialog.Builder dialogBuilder = new AlertDialog.Builder(this);
dialogBuilder.setTitle("Wähle ein Getränk")
        .setIcon(R.mipmap.ic_launcher)
        .setItems(items, new OnClickListener() {
            public void onClick(DialogInterface dialog, int itemPos) {
```

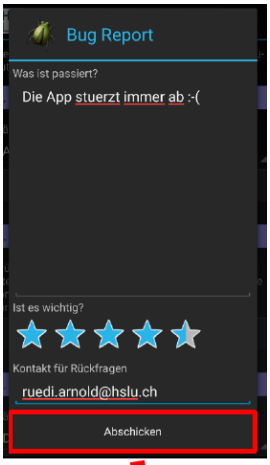
Handler für Selection

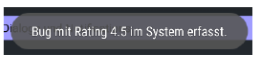
Abbildung 28: Beispiel mit Auswahl-Daten

Alert-Dialog mit eigenem Layout

- Layout.xml „aufblasen“ & setzen








```
private Dialog createCustomLayoutDialog() {
    final View customView = LayoutInflater.from(this).inflate(
            R.layout.custom_dialog_layout, null);
    AlertDialog.Builder dialogBuilder = new AlertDialog.Builder(this);
    dialogBuilder.setTitle("Bug Report").setIcon(R.drawable.ic_bug)
            .setView(customView)
            .setPositiveButton("Abschicken", new OnClickListener() {
                public void onClick(DialogInterface dialog, int which) {
```

Abbildung 29: Beispiel mit eigenem Layout

Ein (offener) Dialog gehört zum Zustand einer Activity, ist ein Dialog noch geöffnet bei einem Konfigurationswechsel, dann wird dieser nicht gespeichert und auch nicht wiederhergestellt! Deshalb sollten Dialoge als `DialogFragment` implementiert werden. Der Zustand des Dialogs wird dann vom `FragmentManager` korrekt mit Lifecycle und Activity synchronisiert (save/restore)
Für den Moment: Ein **Fragment** ist ein wiederverwendbarer „UI Schnippsel“ mit eigenem Zustand und Lifecycle.

2.8.3 Notifications (Status-Bar)

- Persistente Nachricht
  - Kurze Ticker-Nachricht in der Status-Bar
  - Danach persistente Anzeige im Notification Window
  - Bei Auswahl erfolgt Aufruf einer definierten Activity
- Vorteile:
  Nachricht bleibt erhalten bis vom Nutzer quittiert
  Beliebig komplexe Behandlung, da Start einer Activity
- Nachteil:
  Etwas komplexere Mechanik wegen `PendingIntent`

Abbildung 30: Übungs-Demo aus der Vorlesung SW02 - Notification

3 Android 3 - Persistenz & Content Providers

Persistenz: Daten über Laufzeit der App erhalten. Für lokale Persistenz gibt es drei Möglichkeiten:

- **Shared Preferences**
  Key/Value-Paare, Verwendung für kleine Datenmengen
- **Dateisystem**
  intern oder extern, in App-Sandbox (privat) oder auf SD-Karte (öffentlich), Verwendung für binäre/grosse Dateien, Export
- **Datenbank (Room)**
  SQLite + Object Relational Mapper (ORM), Verwendung für strukturierte Daten + Abfragen/Suche

3.1 (Shared) Preferences

- Jede Activity hat ein SharedPreferences-Profil, persistente Einstellungen für Activity oder Applikation
- Key-Value-Store (persistente Map)
- Preferences für **Activity**:
  `Activity.getPreferences(mode)`
  Anwendungsfall: Activity-State persistent speichern
- Preferences für **Applikation**:
  `PreferenceManager.getDefaultSharedPreferences(ctx)`
  `Context.getSharedPreferences(name, mode)`
- Mögliche Datentypen für Preferences-Werte:
  String, int, float, long, boolean, Set<String> (mit seperaten Werten)

Lesen und Schreiben auf Preferences

- Mehrere Dateien pro Applikation möglich, Zugriff mit
  `Activity.getSharedPreferences(name, mode)` (unterschiedliche Dateinamen)
  oder auch über `getDefaultSharedPreferences(mode)`, die Applikation findet danach anhand der Preference-Benennungen die Einträge auch selber
- Lesen mit Methoden `SharedPreferences.get`**X**`()`
  X steht für den Typ, also String, Int, Boolean, ...
- Schreiben immer mit dem Editor:
  1. `SharedPreferences.Editor editor = preferences.edit()`
  2. `editor.put`**X**`(...)`
  3. `editor.apply()` Persistierung der Änderungen
     - asynchrone Persistierung, blockiert die Methode nicht
     - für synchrone Persistierung: `editor.commit()`

Beispiel, um die Anzahl Aufrufe einer App über die Lebenszeit der App hinaus zu persistieren:

```
final SharedPreferences preferences = getPreferences(MODE_PRIVATE);
final int newResumeCount = preferences.getInt(COUNTER_KEY, 0) + 1;
final SharedPreferences.Editor editor = preferences.edit();
editor.putInt(COUNTER_KEY, newResumeCount);
editor.apply();
```

3.1.1 Darstellung User-Preferences

- Automatische Darstellung mit `PreferenceFragment`, eigener Editor für jeden Wertetyp
- `PreferenceFragment` schreibt/liest grundsätzlich in die DefaultSharedPreferences, kann aber auch für andere Preference-Stores konfiguriert werden

User-Präferenzen können in XML deklariert werden unter `res/xml` z.Bsp. als `preferences.xml`, wobei untersch. Präferenzen bspw. als `CheckBoxPreference, ListPreference` usw. erfasst werden. Daten können wie in diesem Beispiel aus den Array-Ressourcen bezogen werden:

```xml
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
    <PreferenceCategory
        android:key="teaPrefs"
        android:title="Tee Präferenzen">

        <CheckBoxPreference
            android:key="teaWithSugar"
            android:persistent="true"
            android:summary="Soll der Tee gesüsst werden?"
            android:title="Gesüsster Tee?" />

        <ListPreference
            android:dependency="teaWithSugar"
            android:entries="@array/teaSweetener"
            android:entryValues="@array/teaSweetenerValues"
            android:key="teaSweetener"
            android:persistent="true"
            android:shouldDisableView="true"
            android:summary="Womit soll der Tee gesüsst werden?"
            android:title="Süssstoff" />

        <EditTextPreference
            android:key="teaPreferred"
            android:persistent="true"
            android:summary="z.B. &quot;Lipton/Pfefferminztee&quot;"
            android:title="Bevorzugte Marke/Sorte" />
    </PreferenceCategory>
</PreferenceScreen>
```

Abbildung 31: Beispiel eines Präferenzen-XML

- Ohne `android:summary` würde die gewählte Preference angezeigt werden
- `android:dependency` deklariert eine Abhängigkeit zu einer anderen Preference, ist diese nicht gegeben kann die andere Preference nicht ausgewählt werden
- **Entries**: „Anzeigestring“, übersetzbar
  EntryValues: „Werte“, nicht übersetzt, technischer Schlüssel

```java
// Zur "Uebersetzung" von Values zu Entries (Beispiel)
public String getValueFromKey(String key) {
  String[] keys = getResources().getStringArray(R.array.teaSweetenerValues);
  String[] values = getResources().getStringArray(R.array.teaSweetener);
  int i = 0;
  while(i < keys.length) {
    if(keys[i].equals(key)) {
      return values[i];
      }
    i++;
  }
  return "";
}
```

3.1.2 PreferenceFragment

Ein `PreferenceFragment` kann in einer eigenen Activity (hier `TeaPreferenceActivity`) erstellt werden:

```
public class TeaPreferenceActivity extends Activity {

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    getFragmentManager().beginTransaction().replace(android.R.id.content,
      new TeaPreferenceInitializer()).commit();
  }

  // PreferenceFragment als statische innere Klasse
  public static final class TeaPreferenceInitializer extends PreferenceFragment
      {
    @Override
    public void onCreate(final Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      addPreferencesFromResource(R.xml.preferences);
      // Referenz auf preferences.xml
  }
}
```

3.1.3 Default-Präferenzen

Präferenzen können programmatisch auch wieder auf „Standard“-Werte oder auf festgelegte Werte gesetzt werden, für das Tee-Beispiel kann dies bspw. folgendermassen vorgenommen werden:

```
SharedPreferences teaPrefs = PreferenceManager.getDefaultSharedPreferences(this);
SharedPreferences.Editor editor = teaPrefs.edit();
editor.putString("teaPreferred", "Lipton/Pfefferminztee");
editor.putString("teaSweetener", "natural");
editor.putBoolean("teaWithSugar", true);
editor.apply();
```

3.2 Dateisystem

- **Einsatzbereiche**
  - Speichern/Laden von binären Dateien (Bilder, Musik, Video, Java-Objects, etc.)
  - Caching (Heruntergeladene Dateien)
  - Grosse Text-Dateien(Plain Text, Strukturierte Daten wie XML, JSON,))
- Teilen / Freigeben von erstelltem Inhalt (Externer Speicher wie SD-Karte)
- Dateien sind entweder
  - PRIVATE → ins Applikationsverzeichnis
    (Zugriff für andere Apps nur über Content Provider möglich)
    * `Context.getFilesDir()`
  - PUBLIC → auf die SD-Karte
    * `Environment.getExternalStorageDirectory()`
      `Environment.getExternalStorageState();`
- Für Zugriff auf SD-Karte muss die Permission im Manifest eingetragen werden! (siehe nachfolgend)

3.2.1 Exkurs: Permission-Model

- Vor gewissen Operationen müssen Apps die Berechtigung des Nutzers erhalten (Kontaktzugriff, Internet, SD-Karte, Kamera, SMS, etc.)
- Klasse: `android.Manifest.permission`
- Seit API 23 werden keine dangerous Permissions mehr gewährt, der Nutzer muss diese selber freigeben (Applikation fragt beim Nutzer nach), Permissions werden einzeln gewährt/abgelehnt.
  Konsequenz: Apps müssen mit eingeschränkten Permissions umgehen können
- Arten von Permissions
  - *normal*
    * Wird bei der Installation automatisch erlaubt
  - *dangerous*
    * Muss von User erlaubt werden (kann wieder entzogen werden)
  - *signature*
    * Wird automatisch erlaubt, wenn die App, welche die Permission definiert, vom gleichen Hersteller ist wie die App, welche die Permission beanträgt (sonst ist sie „dangerous“)
  - *signatureOrSystem*
    * Wird automatisch erlaubt für Apps, welche im System-Image sind, sonst wie „signature“
- Permissions können gruppiert werden, User gibt Freigabe für alle Permissions in einer Gruppe (keine einzelnen Permissions), falls benötigt

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest
    package="ch.hslu.mobpro.persistency"
    xmlns:android="http://schemas.android.com/apk/res/android">

    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.READ_SMS" />
    <uses-permission android:name="android.permission.WRITE_SMS" />
```

Abbildung 32: Erfassung von Permissions im Manifest

3.2.2 Exkurs ff: Runtime Permissions

```
public void loadExtFileWithPermission() {
    int grant = checkSelfPermission(Manifest.permission.READ_EXTERNAL_STORAGE);
    if (grant != PackageManager.PERMISSION_GRANTED) {
        requestPermissions(new String[]{ Manifest.permission.READ_EXTERNAL_STORAGE }, 24);
    } else {
        // permission already granted
        readFile();
    }
}
```

Abbildung 33: RuntimeCheck der Permissions

```
@Override
public void onRequestPermissionsResult(int requestCode, String[] permissions, int[] grantResults) {
    switch (requestCode) {
        case 24: // load file
            if (grantResults.length > 0 && grantResults[0] != PackageManager.PERMISSION_GRANTED) {
                Toast.makeText(this, "Permission " + permissions[0] + " denied!", Toast.LENGTH_SHORT).show();
            } else {
                // permission was granted
                loadFile();
            }
            break;
    }
}
```

Abbildung 34: Callback aus Permission-Abfrage

3.2.3 Exkurs ff: Persistenz mit Datei

Repetition zu Streams, Reader & Co.

- Stream: Byte-Datenstrom `[28,11,200,255,2,15,33]`
  - Auf `File` öffnen:
    `FileOutputStream, FileInputStream`
- Stream kann in Zeichenstrom `['h','a','l','l','o']` umgewandelt werden
  - `FileReader, FileWriter` + „Buffered“-Versionen
- Immer schliessen!
  `stream.close() / reader.close()`
- Nicht vergessen: `try-catch-finally` implementieren
- `java.nio.file.Path`: ist ab API 26 in Android verfügbar!

```
Writer writer = null;
try {
    writer = new BufferedWriter(new FileWriter(outFile));
    writer.write(text);
    return true;
} catch (final IOException ex) {
    // ...
} finally {
    Log.e("HSLU-MobPro-Persistenz", "Got a problem");
    // ...
}
```

Abbildung 35: Beispielcode zur Persistierung in einem Textfile

3.3 Datenbank (Room)

Android-DB **SQLite** ist bei Android fix integriert. Room ist ein Object Relational Mapper (ORM) für Android

3.4 Content Providers